Operating Systems for Low-End Devices in the Internet of Things: a Survey

Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes

***Abstract*—The Internet of Things (IoT) is projected to soon interconnect tens of billions of new devices, in large part also connected to the Internet. IoT devices include both high-end devices which can use traditional go-to operating systems (OS) such as Linux, and low-end devices which cannot, due to stringent resource constraints, e.g. very limited memory, computational power, and power supply. However, large-scale IoT software development, deployment, and maintenance requires an appropriate OS to build upon. In this paper, we thus analyse in detail the specific requirements that an OS should satisfy to run on low-end IoT devices, and we survey applicable operating systems, focusing on candidates that could become an equivalent of Linux for such devices i.e. a one-size-fits-most, open source OS for low-end IoT devices.**

I. Introduction

The Internet of Things (IoT) stems from the availability of a plethora of cheap, tiny, energy-efficient communicating devices (a.k.a. *things*). Multiple standard communication protocols have been developed at different layers for the IoT networking stack, with IPv6 typically being the narrow waist at the network layer. The availability of such protocols enables heterogeneous devices to be interconnected, and reachable from the Internet.

From the hardware point of view, the Internet of Things is composed of heterogeneous hardware - even more than in the traditional Internet. IoT devices can be classified in two categories, based on their capability and performance. The first category consists in *high-end IoT devices*, which includes single-board computers such as the Rasberry Pi [1], and smartphones. High-end IoT devices have enough resources and adequate characteristics to run software based on traditional Operating Systems (OSs) such as Linux or BSD.

The second category consists in *low-end IoT devices*, which are too resource-constrained to run these traditional OSs. Popular examples of low-end IoT devices include Arduino [2], Econotag [3], Zolertia Z1 [4], IoT-LAB M3 nodes [5], OpenMote nodes [6], and TelosB motes [7], some of which are shown in Fig. 1. In this paper, we focus on such low-end IoT devices because they pose novel challenges for OS designers when it comes to handling the highly constrained hardware resources.

A. Low-End IoT Devices

Low-end IoT devices are typically very constrained in terms of resources including energy, CPU, and memory capacity. Recently, the Internet Engineering Task Force (IETF) standardized a classification [8] of such devices in three subcategories[1] based on memory capacity[2].

- *Class 0* devices have the smallest resources (<<10 kB of RAM and <<100 kB Flash); e.g., a specialized mote in a Wireless Sensor Network (WSN).
- *Class 1* devices have medium-level resources (∼10 kB of RAM and ∼100 kB Flash), allowing richer applications and more advanced features than rudimentary motes, e.g. routing and secure communication protocols.
- *Class 2* devices have more resources, but are still very constrained compared to high-end IoT devices and traditional Internet hosts.

On Class 0 devices, extreme specialization and resource constraints typically make the use of a proper OS unsuitable. Therefore, the software running on such hardware is typically developed bare-metal, and very hardware-specific.

IoT devices of Class 1 and above, however, are typically less specialized. Software can alternatively transform such a device into an Internet router [9], host, or server, with a standard network stack and reprogrammable/interchangeable applications running on top of this stack [10]. Therefore, new business models currently emerge based (partly) on portable, hardware-independent software and applications running on IoT devices of Class 1 and above. Consequently, several major companies have recently announced new OSs designed specifically to run on IoT devices, including Huawei [11], ARM [12], and Google [13]. Indeed, on such hardware, it is often desirable to be provided with software primitives enabling easy hardware-independent code production. More generally, there is a need for Application Programming Interfaces (APIs) beyond bare-metal programming that can cater for the wide range of IoT use cases, to facilitate large-scale software development, deployment and maintenance. Such software primitives are typically provided by an OS. In this paper, we will thus focus on OSs that are appropriate for Class 1 and Class 2 devices.

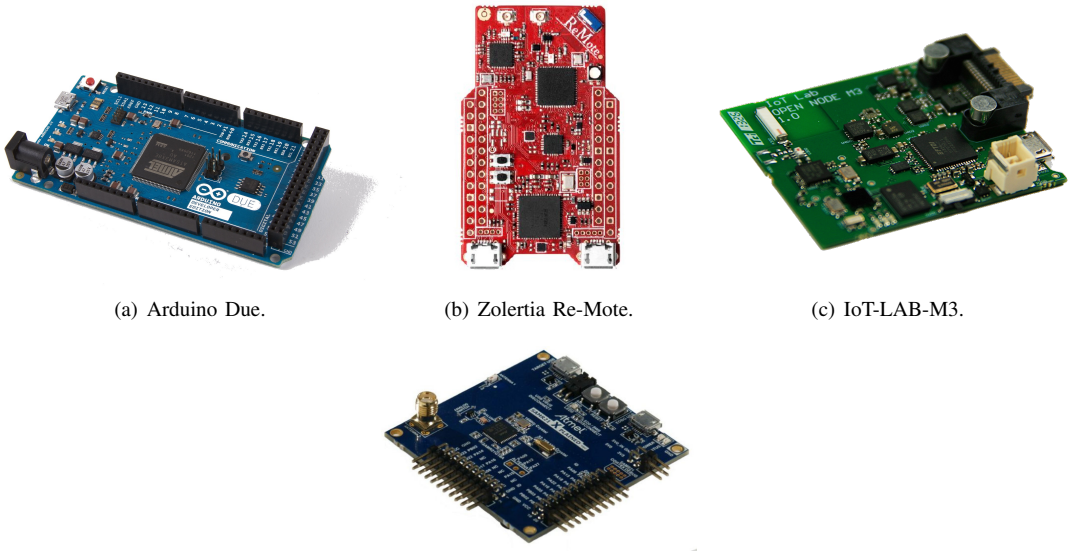
We note that, unfortunately, Moore's law is not expected to help in this context: it is anticipated that IoT devices will get smaller, cheaper, and more energy-efficient, instead of providing significantly more memory or CPU power [14]. Therefore, in the foreseeable future, low-end IoT devices with a few kilobytes of memory, such as Class 1 and Class 2 devices, are likely to remain predominant in the IoT.

O. Hahm and E. Baccelli work for INRIA, France.
H. Petersen works for Freie Universität Berlin, Germany.
N. Tsiftes works for SICS, Sweden.

.

[1]Note that this classification is not to be confused with Electronic Product Code (EPC). It is based on IETF standard classification as specified in RFC 7228 [8]. The terms *Class 0–2* are used according to this classification throughout the paper.

[2]Other classifications, e.g. based on energy capacities, are possible, but available memory is most crucial for the OS design.



(a) Arduino Due. (b) Zolertia Re-Mote. (c) IoT-LAB-M3.

(d) Atmel SAM R21.

Fig. 1: Examples of low-end IoT devices.

B. Operating Systems for Low-End IoT Devices

As previously mentioned, traditional operating systems such as Linux or BSD are not applicable on low-end IoT devices, because they cannot run on the limited resources provided on such hardware. In consequence, the IoT is plagued with lack of interoperability between many incompatible vertical silo solutions. We argue that the IoT will not fulfill its potential until a software big-bang happens, resulting in the emergence of a couple of de facto standard OSs providing consistent API & SDK across heterogeneous IoT hardware platforms.

In this paper, we will thus survey OSs that could become the de facto standard OS for low-end IoT devices. We note that solutions providing the smallest possible memory footprint are typically limited to a specific use case, and are therefore unfit for becoming the generic OS for IoT devices. In contrast, we will thus target one-size-fits-all (or at least one-size-fits-most) solutions that provide the best level of comfort while satisfying medium memory requirements in the order of $\sim$10 kB of RAM or more, and $\sim$100 kB Flash or more; i.e., devices of *Class 1* and above, according to the IETF classification [8].

By level of comfort, we mean interoperability with the rest of the Internet including (i) compatibility with IP protocols from a network point of view, and (ii) from a systems point of view, compatibility with standard programming tools, models, and languages used on Internet hosts. In this paper, we focus on open source OSs, but we will also briefly survey closed source alternatives. One reason for this focus is that several of the most widespread OSs for low-end IoT devices are open source, and that they offer greater possibilities to examine their design and implementation at a thorough level, as is required for this survey. A number of additional reasons for focussing on open source will also be mentioned later in the paper.

The remainder of this paper is organized as follows. First, we analyze the requirements which should be fulfilled by an OS for IoT devices. Then, we overview the main OS design choices and other non-technical factors in this context. Once this background settled, we survey the OSs that are potentially applicable, with the goal of being exhaustive, but brief. Then, we propose a taxonomy for IoT OSs, and we analyse in more depth one OS per identified category, chosen for being prominent within its category.

II. Requirements for an IoT Operating System

In this section we give an overview of the diverse requirements a generic OS for low-end IoT devices should aim to satisfy.

A. Small Memory Footprint

Compared to other connected machines, IoT devices are much more resource-constrained, especially in terms of memory. One of the requirements for a generic OS for the IoT is thus to fit within such memory constraints. While PCs, smartphones, tablets, or laptops provide Giga- or TeraBytes of memory, IoT devices typically provide a few kilobytes of memory, i.e. a million times less. This observation holds both for volatile (RAM) and persistent (ROM) memory [8]. In order to fit within memory footprint constraints, IoT application designers must be provided with a set of optimized libraries (potentially cross-layer) providing common IoT functionality, and efficient data structures.

Identifying the right trade-off between (i) performance, (ii) a convenient API, and (iii) a small OS memory footprint, is a non-trivial challenge. For example, in many cases the OS designer has to identify the sweet spot between RAM and ROM usage. Furthermore, balance must be found between sensible programming guidelines and coding conventions which must be observed on one hand, and the high degree of modularity and configurability which is desired to fit a wide range of use cases on the other hand.

B. Support for Heterogeneous Hardware

While the diversity of hardware and protocols used in today's Internet is relatively small from an architectural perspective, the degree of heterogeneity explodes in the IoT. The large variety of use cases [15]–[19] led to the development of a large variety of hardware and communication technologies. IoT devices are based on various microcontroller (MCU) architectures and families, including 8 bit (e.g. Intel 8051/52, Atmel AVR), 16 bit (e.g. TI MSP430), 32 bit (ARM7, ARM Cortex-M, MIPS32, and even x86) architectures—64 bit architectures might also appear in the future. On top of that, key system characteristics vary wildly: for example some IoT devices provide hundreds of kilobytes of RAM, but no persistent memory to store executable code (and thus generate the need to load both code *and* data into RAM). One such board is the still popular Redwire Econotag board, which is based on an Freescale MC13224V [3], [20]. Other IoT devices are very limited in terms of RAM, but equipped with a lot of ROM, such as the STM32F100VC ARM Cortex-M3 MCU [21]. Similarly, IoT devices can be equipped with a wide variety of communication technologies, as described below in Subsection II-C. Note that such heterogeneity may even occur within a single deployment, whereby many different types of devices take part in various tasks to achieve an overall goal [22], [23]. Thus, one of the requirements—and a key challenge—for a generic OS for the IoT is to support this heterogeneity in hardware architectures and communication technologies.

C. Network Connectivity

The main point of having IoT devices, is that they can interconnect, and communicate with one another or with the Internet. IoT devices are thus typically equipped with one (or more) network interfaces. Communication techniques used in the IoT encompass not only a wide variety of low-power radio technologies (e.g., IEEE 802.15.4, Bluetooth/BLE, DASH7, and EnOcean) but also various wired technologies (e.g., PLC, Ethernet, or several bus systems). Contrary to WSN scenarios [24] [25], it is generally expected that IoT devices seamlessly integrate with the Internet; i.e., can communicate end-to-end with other machines on the Internet [23]. The combination of (i) having to support multiple link layer technologies and (ii) having to communicate with other Internet hosts, led to the use of network stacks based on IP protocols directly on IoT devices [26]. A key requirement for a generic OS for the IoT is thus to support heterogeneous link layer technologies and a network stack based on IP protocols relevant for the IoT [26]. Furthermore, as indicated by the evolution of Linux over the years (which is an obvious example of future-proof design), it is also desirable that the OS can cater for multiple network stacks and for continuous network stack evolution.

D. Energy Efficiency

Many IoT devices will run on batteries or other constrained energy sources. For example, smart meters and other home/building automation devices are required to work for years with a single battery charge [27]. On a global level, energy efficiency is also required due to the sheer number of IoT devices that is expected to be deployed (tens of billions). IoT hardware in general—MCUs, radio transceivers, sensors—provides features to operate in an energy efficient manner. However, there is no free lunch: this yields requirements on IoT software. Indeed, unless IoT software makes use of these features (e.g., putting devices into the deepest sleep mode as often as possible), energy efficiency is not achieved. Therefore, a key requirement for OSs for the IoT is (i) to provide energy saving options to upper layers, and (ii) to make use of these functions itself as much as possible, for example by using techniques such as radio duty cycling, or by minimizing the number of periodic tasks that need to be executed. For instance, a periodic system timer that schedulers use for time slicing leads to a system that never goes to deep power-down modes, and should thus be avoided if possible.

E. Real-Time Capabilities

Precise timing, and timely execution are crucial in various IoT use-cases e.g., smart health applications such as body area networks (BAN) with pacemakers providing wireless monitoring and control [28], [29], or in other scenarios including actuators and/or robots in industrial automation contexts, or a Vehicular Ad-Hoc Network (VANET). An OS that can fulfill timely execution requirements is called a Real-Time Operating System (RTOS), and is designed to guarantee worst-case execution times and worst-case interrupt latencies. Therefore, another requirement for a generic OS for the IoT is to be an RTOS, which typically implies that kernel functions have to operate with a deterministic run-time. The Japanese open standard for a real-time operating system, ITRON, is popular in this field, though it aims mostly for consumer electronics [30].

F. Security

On one hand, some IoT systems are part of critical infrastructure or industrial systems with life safety implications [31]. On the other hand, since they are connected to the Internet, IoT devices are in general expected to meet high security and privacy standards. Beyond the overarching trust management challenge, IoT security challenges includes data integrity, authentication, and access control in various parts of the IoT architecture. Thus, a requirement (and challenge) for an OS for the IoT is to provide the necessary mechanisms (cryptographic libraries and security protocols) while retaining flexibility and usability. Last but not least, since software with a certain degree of complexity can never be expected to be 100% bug-free, and security standards evolve (driven by various stake holders such as industry, government, consumers etc.) it is crucial to provide mechanisms for software updates on already-deployed IoT devices—and to use open source as much as possible [32].

III. Key Design Choices

The success and applicability of an OS for the IoT are influenced by technical as well as political or organizational

factors. In this section, we will overview key technical OS design alternatives, as well as relevant non-technical considerations.

A. Technical Properties

Design choices concerning, e.g., the general OS model, the scheduling strategy, or hardware abstraction, have a major impact on the capabilities and flexibility of the system. In this section, we will overview such choices and how they affect OS applicability for IoT use cases.

General Architecture and Modularity. The first design decision that has to be made for any OS is the choice of the kernel type. This choice has a major impact on the overall architecture of the system and its modularity. A generic architecture for an IoT OS is depicted in Figure 2. One can differentiate between an *exokernel* approach, a *microkernel* approach, a *monolithic* approach, or a hybrid approach. The main idea behind the exokernel approach is to put as few abstractions as possible between the application and the hardware, and to mostly focus on avoiding resource conflicts and checking access levels. The microkernel approach aims for more functionalities (minimalistic set of features) in the kernel, while still requiring very little memory, and providing a lot of space and flexibility for the rest of the system, as well as robustness (since a crashing device driver will not affect the stability of the whole system). However, due to the typical absence of an Memory Management Unit (MMU) on low-end IoT devices, buffer and stack overflows can still happen and have severe impact on the system. Finally, the main idea behind a monolithic approach is that all components of the system are developed together, which may lead to a simpler and overall more efficient design.

Synopsis: One has to choose between the more robust and more flexible microkernel or a less complex and more efficient monolithic kernel — or go for a hybrid approach.

Scheduling Model. Another crucial part of any OS is the scheduler, which affects other important properties such as energy efficiency, real-time capabilities, or the programming model. There are typically two types of schedulers: preemptive schedulers, and non-preemptive (or cooperative) schedulers. An OS may provide different schedulers, that can be selected at build time. A preemptive scheduler can interrupt any (non-kernel) task at any given point to allow another task to execute for a limited time. In a cooperative model, each thread is responsible to yield itself, because no other task, and in some cases not even the kernel, is able to interrupt a task.

In many cases a preemptive scheduler requires a periodic timer tick, sometimes called a *systick*, in order to assign time slices to each task. This requirement usually prevents the IoT device to enter the deepest power-save mode, since at least one hardware timer needs to stay active. Additionally, the MCU enters full active mode at each *systick*. Time-sliced scheduling is often used for OSs with a User Interface (UI) to mimic a parallelized execution of multiple tasks. For IoT OSs this is mostly unnecessary because they do not have a direct user and, thus, do not require a UI.

Synopsis: A preemptive scheduler assigns CPU time to each task, while the different tasks have to yield themselves in the cooperative model.

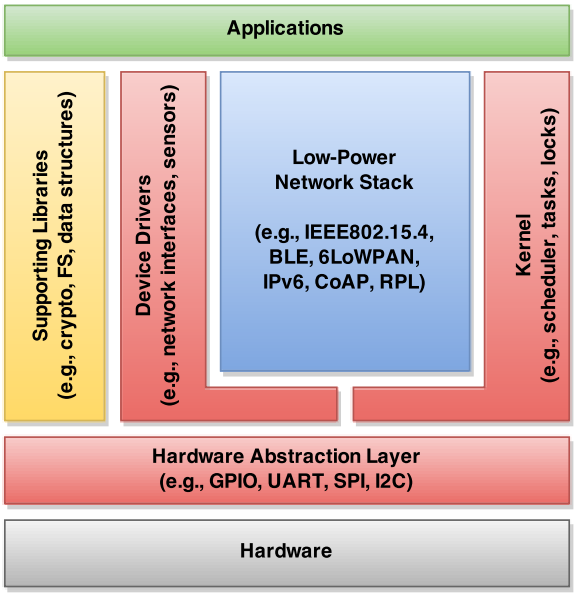


Fig. 2: Typical components of an OS for low-end IoT devices, including a common low-power IPv6 protocol stack.

Memory Allocation. As described in Section II, memory is usually a very scarce resource on IoT devices. Hence, a sophisticated handling of memory is required. One important question is whether memory is allocated in a static or dynamic manner, and this choice also affects other criteria of the system design. Static memory allocation typically requires some over-provisioning and makes the system less flexible to changing requirements during run-time. Dynamic memory allocation makes the system design more complicated for two main reasons. First, functions such as `malloc()` and related functions are usually implemented in a time-wise non-deterministic fashion in the standard C libraries and, thus, will break any real-time guarantees. Hence, in order to make use of dynamic memory allocation for applications with real-time requirements, the OS has to provide special implementations for deterministic `malloc()` like TLSF [33]. Second, dynamic memory allocation creates the need to handle out-of-memory situations and the like at runtime, which may be difficult to deal with. Additionally, heap-based malloc implementations usually induce memory fragmentation, which cause systems to run out of memory even faster.

Synopsis: Static memory allocation introduces some memory overhead due to over-provisioning and results in less flexible systems, while dynamic memory allocation leads to a more complex system and may conflict with real-time requirements.

Network Buffer Management. A central component of an IoT OS is the network stack where chunks of memory, e.g., packets, has to be shared between the layers. Two possible solutions to achieve this are copying of memory (`memcpy()`) or passing of pointers between the several layers. While the first solution is expensive from a resource point of view, the

latter generates the question who is responsible to allocate the memory. Delegating this task to the upper layers, make the application development more complex and less convenient. Leaving this task for the lower layers, such as the device driver, make the system less flexible. A possible approach to solve this conflict is the design of a central memory manager as proposed for TinyOS or RIOT [34], [35].

Synopsis: Memory for packet handling in the network stack may be allocated by each layer or passed as a reference between the layers.

Programming Model. The *programming model* defines how an application developer can model the program. The typical *programming models* in the domain of IoT OSs can be divided into event-driven systems and multi-threaded systems. In an event-driven system which is, for example, widely used for WSN OSs, every task has to be triggered by an (external) event, such as an interrupt. This approach is often accompanied by a simple event loop (instead of a more complex scheduler) and a shared-stack model. A programming model based on multi-threading gives the developer the opportunity to run each task in its own thread context, and communicate between the tasks by using an Inter Process Communication (IPC) API.

Synopsis: Event-driven systems can be more memory-efficient, while multi-threading systems eases the application design.

Programming Languages. The main choice for the programming language of an OS is to decide between (i) a standard programming language, typically ANSI C or C++, and (ii) an OS-specific language or dialect. On the one hand, providing OS-specific language features allows performance- or safety-relevant enhancements that low level languages like C do not support. On the other hand, they prevent the use of well-established and mature development tools. The specification of standards for programming languages, most notably the ANSI specifications for C and C++, meant a significant boost for the evolution of software in general and for OSs in particular. Despite its age (and the rise of newer programming languages), the C programming language is still the most important and most widely used programming language (along with Assembler) when it comes to OS programming, and to lower level parts such as scheduling or device drivers. However, more sophisticated languages with a bigger feature set may be available on top of that, at higher levels, to ease application programming.

Synopsis: Standard programming languages simplify portability and enable the use of well-known development tools. OS-specific languages and language extensions can increase the system performance and safety.

Driver Model and Hardware Abstraction Layer. IoT systems will interact with the environment in many ways, either in a passive way by sensing through all kind of sensors or actively through actuators such as motors or lighting systems. Consequently, MCUs for these systems are usually equipped with a variety of different peripheral devices, like ADCs/DACs, interfaces like SPI, I^2C, CAN bus, or serial lines, and GPIOs. Thus, a flexible and reasonably convenient driver interface is crucial for an IoT OS.

In addition to the driver model for connecting external devices, e.g., sensors, actuators, transceivers, the model may also abstract from the underlying hardware in general. A hardware abstraction layer can provide a well-defined interface to CPU, memory, and interrupt handling in order to make porting to new platforms a straightforward task.

Synopsis: A well-defined hardware abstraction layer and driver model can significantly improve the system design, but introduces a certain amount of overhead—either in terms of lines of code or in terms of runtime overhead.

Debugging Tools. As mentioned before, the choice of programming languages also predetermines the possible tools to use, including the ones for debugging. Well-established toolchains such as the one around the GNU Compiler Collection (GCC) usually include corresponding debugging tools, e.g., the GNU Debugger (GDB). However, in order to run a live debugging system, the target board has to provide an adequate interface, such as JTAG or Spy-Bi-Wire. Unfortunately, not every IoT device provides such an interface, and therefore other debugging facilities are needed.

A common auxiliary tool is the use of `printf()` and the like for simple debugging over a serial interface, e.g., a USART. In some cases, even a simple LED blinking algorithm can sometimes be found as a primitive debugging substitute. If one lacks access to the devices, as is often the case with deployed IoT networks, it is necessary to provide other means for accessing debug information. For instance, this can be achieved through periodic diagnostic messages sent over the network, or through logs written on external flash memory.

Synopsis: Using standard programming languages in general allows for using standard debugging tools, but hardware limitations may pose the need for other, simpler debugging facilities via serial output or even LED blinking.

Feature Set. An OS can be split into kernel and higher level functionalities. Typically the kernel provides a scheduler, a model for tasks, mutual exclusion (mutex) and other forms of synchronization, and timers. In case the OS supports multi-threading, the API will usually also comprise functions for IPC. On higher layers, system libraries can be found, such as a shell, logging, cryptographic functions, or network stacks. Due to typically missing MMUs on IoT devices, such applications and application libraries will usually run in the same address space as kernel operations and can therefore decrease the system's stability.

In addition to network protocols, features in higher layers that are of particular interest in an OS for low-end IoT devices include over-the-air updates, dynamic loading and linking, or libraries for lightweight encryption and decryption.

Synopsis: The overall feature set of an OS may be described by the size of its API.

Testing. As for all software systems, testing plays a crucial role for the development of IoT OSs. In particular, for highly distributed development workflows, as can often be found in bigger open source projects, deploying a continuous integration (CI) environment is inevitable [36]. This CI will usually include build and integration tests as well as unit and regression tests. The specific challenges of testing for IoT systems arise from the distributed nature of these systems, and the fact that they are deeply embedded and often very constrained. A

widely used approach to deal with the hardware-related part of the testing, such as the testing of device drivers, is to use hardware emulation tools, e.g., MSPSim or Emul8 [37], [38]. Network emulators and simulators such as Cooja or ns-2/ns-3, that allow for the integration of OS code, are of great help in this context [39].

Synopsis: The distributed nature and constraints of the hardware makes thorough testing a challenging, but crucial task.

B. Non-Technical Properties

The applicability of a technically fit OS—in particular for commercial usage—is also influenced by aspects such as the license, maintainability, the workflow, or the provider of the OS. In this section we overview such non-technical aspects.

(Open) Standards. A crucial characteristic for any OS is its ability to provide applications portability across hardware platforms and architectures—ideally, without any additional effort. Standardized APIs (such as POSIX, specified by IEEE and the Open Group) were also developed to simplify software porting between several OSs. However, on low-end IoT devices, implementing a standard API designed for general purpose operating systems such as Linux may be difficult because of software size constraints (and in fact, even on PCs, few OSs can claim full POSIX compliance). For seamless software porting between multiple OSs, additional support for programming language standards such as ANSI C99 or C++11 should nevertheless be provided. Finally, standards are not only important on the system level, but unavoidable on the network level. For standards at the network level, experience shows that the use of open-access specifications, such as those standardized by the Internet Engineering Task Force (IETF) for instance, is preferable by default over other approaches.

Synopsis: The use of standards improves portability and interoperability.

Certification. For some use cases, in particular for critical systems in applications such as building automation, crucial properties of the system include real-time capabilities, robustness, or determinism. In these cases, certification through independent institutions becomes an inevitable requirement for the OS. A typical and widely established example for such a certification is the IEC 61508 standard, which is titled ”Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems”. Additional certifications that are relevant for OSs on IoT devices are the IPv6 Forum’s ”IPv6 Ready” logo program, and the recently started IPSO Alliance compliance and certification program.

Synopsis: Especially for the deployment in industrial and safety-critical applications, certification of the entire software running on an IoT system might be mandatory.

Documentation. Complete and easy-to-understand documentation is important for any piece of software. For an OS this requirement becomes even more important as the OS is the foundation of every other piece of software running on the system. Furthermore, need for thorough documentation is exacerbated for embedded software, as such software often has to make compromises for reasons that are difficult to grasp at first sight, due to constraints that are only partly apparent. A typical indicator for thoroughly documented code (but not necessarily the most meaningful measure) is the percentage of documentation per lines of code.

Synopsis: In order to make the best use of an OS and ease application design, a complete and comprehensible documentation is required.

Maturity of the Code. Even more difficult to measure than the quality of documentation is the maturity of software. A very rough indicator is the age of the project combined with the number of contributors and users. While certification is in many cases mainly a legal safeguard, the actual robustness and correctness of a system is much more difficult to assess.

Synopsis: In many cases thorough testing and wide deployment in commercial applications is a better indicator for the maturity of an OS than the mere age of the project or certifications.

License of the Code. In general, one can distinguish between three license categories: (i) non-free, (ii) permissive open source, and (iii) copyleft licenses. If an OS is released under a non-free license, the OS is either only available as binary data, or customers are charged extra fees to obtain the source code, which hampers bug fixes and improvements by third parties by limiting the number of contributors [32]. Permissive licenses, e.g. BSD, MIT, or Apache License, give developers and users a high degree of freedom, and are often more easily accepted by industry than copyleft licenses—although for some companies, quite the contrary is true. A possible downside of permissive licences is the potential fragmentation of the community and code base, which often leads to a situation where not all features are accessible—or at least not within one repository. By contrast, copyleft licenses such as GPL (with or without linking exception) and LGPL, are less easily accepted by some industry branches, but can lead to a much more integrative community and a common code base, as can be seen with the outstanding example of Linux.

Synopsis: Open source—particularly copyleft licenses—may not always be the first choice of industry, but offers chances for higher code quality and more secure code due to the increased numbers of contributors and reviewers.

Provider of the OS. The code of the OS may be provided in different forms and by differing entities (depending on the chosen license type). It might be either provided by the vendor that actually develops the software, or by a third party, which may also provide commercial support. In case of open source solutions, the code is often provided by the developer community itself through repositories of version control systems such as Git, Subversion, or Mercurial. The community typically provides best-effort support via online forums, open issue trackers, and mailing lists for these type of projects. This support is crucial in practice and it is thus highly recommended to prefer an open source project with a currently active community, over an open source project with no active community, or with a formerly active community. Note that sometimes, professional software consulting is offered not only for commercial OSs, but also for free open source OSs.

Synopsis: The way of distribution and degree of support for

an OS is highly dependent on its license.

IV. Candidate OS for the IoT

In this section, we briefly review OSs that represent the most promising approaches towards a generic IoT OS. The goal in this section is exhaustiveness rather than in-depth analysis (which is the focus of the next section). We will distinguish between (i) open source OSs, (ii) closed source OSs, and (iii) other software libraries or middleware for the IoT. If not mentioned otherwise, all OSs are written in the C programming language, while some hardware-specific parts may be implemented in assembly language.

A. Open Source OSs

This section lists the predominant open source OSs targeting for IoT devices.

1) Contiki [40], [41]: Contiki was originally developed as an OS for WSNs running on very memory-constrained 8-bit MCUs, but now also runs on 16-bit MCUs and modern IoT devices based on the ARM 32-bit MCUs. It is based on an event-driven, cooperative scheduling approach, with support for lightweight pseudo-threading. While being written in the C programming language, some parts of the OS make use of macro-based abstractions (e.g., Protothreads [42]), and in effect require developers to consider certain restrictions as to what type of language features they can use. Contiki code is available under BSD license on GitHub[3] and other platforms, while a large variety of forks are developed independently (including many closed source versions of the OS). Contiki features several network stacks, including the popular uIP stack, with support for IPv6, 6LoWPAN, RPL, and CoAP; and the Rime stack, which provides a set of distributed programming abstractions. Contiki is developed since 2002, and is so far one of the most used open source OSs for constrained nodes.

2) RIOT [43]–[45]: RIOT was developed with the particular requirements of IoT in mind and aims for a developer-friendly programing model and API, e.g. similar to what is experienced on Linux. RIOT is a microkernel-based RTOS with multi-threading support, using an architecture inherited from FireKernel [46]. While the OS is written in C (ANSI99), applications and libraries can also be implemented in C++. The source code is available on GitHub[4] under LGPLv2.1. RIOT features several network stacks, including its own implementation of the full 6LoWPAN stack (the *gnrc* stack), a port of the 6TiSCH stack OpenWSN [47], and a port of the information centric networking stack CCN-lite [48]. RIOT is developed as such since 2012, by a growing, world-wide open source community.

3) FreeRTOS [49]: FreeRTOS is a popular RTOS which has been ported to many MCUs. Its preemptive microkernel has support for multi-threading. It is now developed by Real Time Engineers Ltd. and its code is available on the project page under a modified GPL that allows commercial usage with closed source applications (only the kernel has to remain open source). Although it does not provide its own network stack, third-party network stacks can be used for Internet connectivity. FreeRTOS is developed since 2002, and is so far one of the most used open source RTOSs for constrained nodes.

4) TinyOS [50]: Together with Contiki, TinyOS is the most prominent OS for WSN applications, targeting very constrained 8 bit and 16 bit platforms and is known for its sophisticated design. TinyOS and nesC evolved language primitives and programming abstractions to prevent as many bugs as possible through software structure and enhance memory efficiency by reducing the actual linked code to a minimum. However, the rather complex design in combination with a customized programming language makes it hard to learn, and it is thus lacking a bigger developer community [51]. It follows an event-driven approach, where several *components* or *modules* can be virtually wired, as described by *configurations* according the requirements. It is written in a dialect of the C programming language, called *nesC*. Its source code is available online under the BSD license on GitHub[5]. The included BLIP network stack implements the 6LoWPAN stack. TinyOS is developed since 2000, and is so far one of the most used open source OSs for constrained nodes, with Contiki.

5) OpenWSN [47]: OpenWSN comprises a 6TiSCH network stack, a basic scheduler, and a *Board Support Package (BSP)* i.e., a simple hardware abstraction, making it possible to run OpenWSN on a dozen IoT hardware platforms. As such, OpenWSN is more of a network stack than a full-fledged OS. OpenWSN code is available online under the BSD license on GitHub[6]. The main focus of OpenWSN is the 6TiSCH network stack, including an implementation of the IEEE 802.15.4e MAC amendment [26]. OpenWSN is developed since 2010, by a growing, world-wide open source community.

6) nuttX [52]: The nuttX OS aims for full POSIX and ANSI compliance and supports MCUs ranging from 8 bit up to 32 bit architectures. NuttX can be built as a microkernel as well as a monolithic version. It is highly modular and features real-time capabilities as well as a tickless scheduler. The source code is available under BSD license on Sourceforge[7]. The integrated network stack includes support for IPv4 and IPv6 with various upper layer protocols. NuttX is developed since 2007.

7) eCos [53]: The *embedded configurable operating system* (eCos) supports 16, 32, and 64 bit embedded hardware. eCos code is available under a custom license based on GPL with linking exception (acknowledged by FSF). While the open source version of eCos seems rather inactive, the commercial version (eCosPro by eCosCentric) is under active development. eCos does not provide an own network stack per se, but supports third-party network stacks (lwIP and the FreeBSD network stack). The source code is available in a Mercurial repository[8]. eCos is developed since 2002, but parts of the code-base are older.

[3] see https://github.com/contiki-os/contiki
[4] see https://github.com/RIOT-OS/RIOT
[5] see https://github.com/tinyos/tinyos-main
[6] see https://github.com/openwsn-berkeley/openwsn-fw
[7] see http://git.code.sf.net/p/nuttx/git
[8] see http://hg-pub.ecoscentric.com/ecos/

name	architecture	scheduler	programming model	targeted device class[a]	supported MCU families or vendors	programming languages	license	network stacks
Contiki	monolithic	cooperative	event-driven, Protothreads	Class 0 + 1	AVR, MSP430, ARM7, ARM Cortex-M, PIC32, 6502	C[b]	BSD	uIP, RIME
RIOT	microkernel RTOS	preemptive, tickless	multi-threading	Class 1 + 2	AVR, MSP430, ARM7, ARM Cortex-M, x86	C, C++	LGPLv2	gnrc, OpenWSN, ccn-lite
FreeRTOS	microkernel RTOS	preemptive, optional tickless	multi-threading	Class 1 + 2	AVR, MSP430, ARM, x86, 8052, Renesas[c]	C	modified GPL[d]	None
TinyOS	monolithic	cooperative	event-driven	Class 0	AVR, MSP430, px27ax	nesC	BSD	BLIP
OpenWSN	monolithic	cooperative[e]	event-driven	Class 0 – 2	MSP430, ARM Cortex-M	C	BSD	OpenWSN
nuttX	monolithic or microkernel	preemptive (priority-based or round robin)	multi-threading	Class 1 + 2	AVR, MSP430, ARM7, ARM9, ARM Cortex-M, MIPS32, x86, 8052, Renesas	C	BSD	native
eCos	monolithic RTOS	preemptive	multi-threading	Class 1 + 2	ARM, IA-32, Motorola, MIPS . . .	C	eCos License[f]	lwIP, BSD
uClinux	monolithic	preemptive	multi-threading	>Class 2	Motorola, ARM7, ARM Cortex-M, Atari	C	GPLv2	Linux
ChibiOS/RT	microkernel	preemptive	multi-threading	Class 1 + 2	AVR, MSP430, ARM Cortex-M	C	Triple License[g]	None
CoOS	microkernel RTOS	preemptive	multi-threading	Class 2	ARM Cortex-M	C	BSD	None
nanoRK	monlothic (resource kernel)	preemptive	multi-threading	Class 0	AVR, MSP430,	C	Dual License	None
Nut/OS	monolithic	cooperative	multi-threading	Class 0 + 1	AVR, ARM	C	BSD	native

TABLE I: Overview of potential open source OSs for the IoT

[a] According to [8]: Class 0 devices have <<10 kB RAM and <<100 kB ROM, Class 1 devices have ∼10 kB and ∼100 kB ROM, Class 2 devices have ∼10 kB and ∼100 kB ROM.
[b] with some restrictions
[c] and several other MCUs
[d] added an exception to allow commercial use
[e] only a rudimentary scheduler is provided, support for RIOT and FreeRTOS scheduler
[f] GPL with linking exception
[g] Developer version under GPL, Releases under GPL with linking exception, commercial licensing is possible

8) mbedOS [12]: ARM recently released a technology preview release (labeled 15.11) of their upcoming OS for low-end IoT devices, called mbed OS. Based on this preview, mbedOS focuses exclusively on 32 bit ARM embedded architecture, and supports a small number of platforms (5 so far, though we can expect many more in the near future). Among the experimental features show-cased in the preview are a (closed-source) 6LoWPAN implementation that claims to implement the *Thread 1.0* specification, several interface definitions, a port of PolarSSL, and support for Bluetooth Low Energy. mbed is developed since 2009, but had so far focused on providing a hardware abstraction layer rather than an OS.

9) L4 microkernel family [54], [55]: L4 OSs follow a strict microkernel design and were originally created to overcome the poor performance of earlier microkernel-based OSs in the mid-1990s. Later implementations have been designed for platform independence, improved security, isolation, and robustness. A well-known representative of this family is seL4, developed in 2006 by the NICTA group with a particular focus on security, reliability, and formal verification [56]. However, most L4 microkernel based OSs do not match the constraints of Class 1 devices. An exception is the F9 microkernel that targets particular ARM Cortex-M3/M4 based devices. While many members of this family are licensed under GPL or BSD license, not all of them are open source.

10) uClinux [57]: This is a port of the Linux 2.x kernel for CPUs without an MMU and with a much smaller memory footprint than Linux. While uClinux benefits from the rich feature set of Linux (including APIs, a full TCP/IP stack, and excellent file system support), it has the drawback of memory requirements that do not fit low-end IoT devices, such as Class 1 devices [8], which are the focus of this survey. The source code is available on Sourceforge[9]. uClinux is developed since 1998.

11) Android [58] and Brillo [59]: This mobile OS Android, developed by Google, is a variant of Linux, targeting mostly smartphones and tablets, but has also been used in cars, watches, TVs, and other consumer electronics. The concept of *apps*, accessible through online stores where users can purchase and download application software, boosted the evolution of smartphones. While the core of Android is open source—as required by Linux' GPL—many of the device drivers and hardware support is proprietary closed source code. Similarly to other Linux-based systems, Android is unable to run on low-end IoT devices such as Class 1 devices.

In 2015, Google announced Brillo [59], a slimmed-down version of Android that will be able to run on IoT devices offering a few tens of megabytes of memory. Hence, Brillo requires considerably less hardware resources than Android. Because it is still a variant of Linux, however, it cannot be used on the low-end IoT devices that are the focus of this survey, and we will therefore not expound its technical details.

12) Other open source OS: For sake of completeness, we mention below other open source OSs. However, since they are not as prominent, we describe them in less detail.

[9]see http://sourceforge.net/projects/uclinux/files/

- *ChibiOS/RT* [60] is an RTOS is developed since 2007 under a modified GPL with linking exception and aims for high performance on 8, 16, and 32 bit MCUs.
- *CooCox CoOS* [61] is a free and open RTOS specially designed for ARM Cortex-M platforms which comes along with a full-fledged IDE, developed since 2009.
- *ERIKA Enterprise* [62] is an RTOS targeted for automotive embedded systems. It supports 8, 16, and 32 bit MCUs, has support for multi-core systems and is licensed under GPL v2 with linking exception.
- *MansOS* [63] is another WSN OS that aims for easy developing and debugging and supports currently 8 bit AVR and 16 bit MSP430 MCUs.
- *NanoQplus* [64] developed at ETRI targets WSN Class 0 devices and provides multi-threading and a memory protection mechanism.
- *nanoRK* [65] is an RTOS for WSNs with a focus on resource reservation for tasks, developed since 2005 for MSP430 platforms.
- *Nut/OS* [66] emerged from an RTOS called *Liquorice* [67], Nut/OS focusses on constrained devices with wired (Ethernet) connections.
- *RTEMS* [68] is an open RTOS with focus on open standard APIs, multiprocessor support, and hard real-time guarantees.
- There are other open source OSs from the domain of WSNs, such as *SOS* [69], *MANTIS OS* [70], [71], *Lorien* [72] or *LiteOS* [73], but they are mostly inactive and never targeted IoT scenarios.

A detailed tabular overview of the open source OS listed above is given in Table I. On the other hand, Table II summarizes why OSs like Contiki, FreeRTOS, or RIOT are a good match to most of the requirements derived in Section II, while other approaches such as uClinux, Arduino and Android fail to fulfill them.

B. Closed Source OSs

In addition to the aforementioned open source OSs, several closed-source OSs have characteristics suitable for IoT domain. Albeit being proprietary, some vendors offer limited access to their source code for customers, registered users, or academic institutes. These OSs, however, are often originally designed for other domains, and typically lack important features such as energy-saving mechanisms or recently standardized IoT protocols. Still, some of the closed-source OSs can be adapted to run on Class 0 and Class 1 devices, and we the list some of the more relevant examples below.

1) ThreadX [74]: ThreadX is an RTOS developed by *Express Logic, Inc.* which has recently been acquired by ARM (and might become the core of mbed OS 3.0) [75]. ThreadX is based on a microkernel RTOS (sometimes referred to as a picokernel) which supports multi-threading and uses a preemptive scheduler. The kernel provides two techniques to eliminate priority inversion: (i) priority inheritance that elevates the priority level of a task while executing a critical section and (ii) preemption threshold that disable preemption of threads below a specified priority. Additional features such

as a network stack, USB support, a file system, or a GUI can be purchased as separate products.

2) QNX [76]: Originally developed by Quantum Software Systems in 1982, QNX was acquired by Research in Motion (RIM) in 2010. It was one of the first commercially successful microkernel-based RTOSs and provides a UNIX-like API. QNX's powerful IPC served as inspiration for many subsequent OSs, such as RIOT. The current version, called *QNX Neutrino*, supports numerous architectures, but none of them matching the requirements of Class 1 devices.

3) VxWorks [77]: Developed initially in 1987 by Wind River (which is now owned by Intel), VxWorks is a monolithic kernel that mostly supports ARM platforms and Intel platforms, including the new Quark SoC. VxWorks supports IPv6 and other IoT features, but lacks support for a 6LoWPAN stack, and cannot fit on constrained IoT devices as defined by RFC 7228 [8] which are the focus of this survey.

4) Wind River Rocket [78]: Another OS developed by Wind River is Rocket which targets particular IoT scenarios. So far, Rocket supports a single hardware platform: Intel's Galileo Gen 2 board which offers several megabytes of RAM and ROM. The OS is tightly bound to using Wind River's cloud platform Helix.

5) PikeOS [79]: PikeOS is developed since 1991 by a company called SYSGO AG (now owned by Thales). PikeOS is a microkernel-based RTOS, which provides safety and security, and acts as a hypervisor for other OSs. Originally called P4, PikeOS is a descendent of the L4 microkernel family. PikeOS provides multiple APIs, can host various guest OSs, and is certified according to several relevant standards including IEC 61508 or EN 50128.

6) embOS [80]: embOS is developed by *Segger Microcontroller Systems*, a company providing development and programming tools as well as software for embedded devices. embOS is an acrtos written in ANSI C, featuring a priority-based, tickless, preemptive scheduler, and targeting various constrained 8 bit, 16 bit, and 32 bit MCUs. A network stack (including ZigBee), USB support, a GUI, and a file system are available as separate add-on products.

7) Nucleus RTOS [81]: Nucleus is an RTOS developed by Mentor Graphics, an electronic design automation company, which acquired the former provider of Nucleus, Accelerated Technology, in 2002. Nucleus enables C++ programming, is POSIX-compliant, and compatible with the Micro ITRON interface. Nucleus has a rich feature set, including an IP network stack, and can be scaled down to tens of kilobyte, but it is not among the RTOSs with the smallest memory footprints, however.

8) Sciopta [82]: Sciopta is an RTOS provided by SCIOPTA Systems AG, with a focus on safety-critical applications. Its microkernel (with a direct message passing IPC) and scheduler are written in assembler. The supported architectures comprise ARM7, ARM9, ARM Cortex-M, ARM Cortex-A, and PowerPC. SCIOPTA Systems also offers additional modules for, e.g. a FAT file system or an IP-based network stack.

9) µC/OS-II and µC/OS-III [83], [84]: µC/OS-II and µC/OS-III are two versions of an RTOS provided by Micrium Inc.. These RTOSs are based on a microkernel with multi-threading and IPC capabilities. In comparison to µC/OS-II, the version released in 2009, µC/OS-III comprises some enhanced features such as unlimited number of tasks and priorities. Additional software packages such as a GUI, a file system, or a TCP/IP network stack are also provided by Micrium, and can be integrated into µC/OS-III.

10) µ-velOSity [85]: µ-velOSity is a royalty-free RTOS developed by Green Hills Software (GHS). Well integrated into Green Hills' IDE (called MULTI), µ-velOSity is written in MISRA-compliant ANSI C and based on a microkernel. Similarly to other commercial IoT OSs, additional required features (e.g., a network stack) are provided separately. Note however that a 6LoWPAN stack is not available.

11) Windows CE [86]: Windows CE is a version of the Windows OS for constrained devices, and has been developed by Microsoft since 1996. Windows CE is real-time capable and has a rich feature set. However, it requires ROM and RAM in the order of megabytes, and therefore targets devices that are less resource-constrained than low-end IoT devices, which are the focus of this survey.

12) LiteOS Huawei [87]: In 2015, Huawei announced [11] that they will release LiteOS, an operating system for IoT devices. The announcement claimed Huawei's LiteOS will fit within 10 kBytes of memory, and will be the most lightweight IoT operating system. For now the code is not available [87] and it is unclear if the OS will indeed be open source, hence we list it in the present category. Furthermore the technical characteristics of this OS are unknown, and in particular, it is unclear how it relates to the open source OS called LiteOS [73] which we mentioned in the previous section.

C. Other Software

For the sake of completeness, we also summarize in this section a collection of other pieces of software that are sometimes mentioned as potential contenders, but in fact are not full-fledged OSs, or are not applicable on Class 1 devices.

1) Arduino [88]: Originating from a university project, Arduino is an open source hardware and software company. Bundled with an IDE targeting people unfamiliar with programming, it enables easy prototyping. Good support for hardware features is achieved by the fact that Arduino provides both platforms and software. Arduino does not, however, provide a real scheduler, support for threading, or any higher layer functionality, thus making it suitable primarily for simpler applications.

2) Espruino [89]: Espruino provides several embedded platforms and an open source software environment. The software part is a very efficient interpreter for JavaScript that makes it feasible to run JavaScript code on constrained devices with less than 100 kB of RAM. However, similar to Arduino, the Espruino does not aim to replace a full-featured OS, but rather to provide a scripting framework for hobbyists and makers. It does not provide basic OS functionality such as a scheduler or thread management. Due to the nature of a scripting language, it is furthermore not capable of fulfilling real-time guarantees or fit on low-end IoT devices, but rather devices such as Tessel [90].

3) node OS [91]: Node OS is a toolset written entirely in Javascript. Although its name suggests it is an OS, node OS is rather a middleware than an OS itself. It does not operate directly on the hardware, but runs on top of the Linux kernel. The requirement for Linux, coupled with the overhead of Javascript, make Node OS inappropriate for low-end IoT devices such as Class 1 devices.

V. Categorization of OS Relevant for IoT

In the following, we will focus on open source OSs. The reasons for this are (i) security and trustworthiness through transparency of code running on IoT devices, and (ii) the anticipated need to spread development costs between multiple parties (similarly to Linux). The open source OSs surveyed in Section IV can be categorized by their architectural concept into three main categories: (i) event-driven OSs, multi-threading OSs, and (iii) pure RTOSs. Although there is some overlap between these categories, they will define the main characteristic of an OS. This section will describe in more details the characteristics of each category, and identify the most prominent, representative OS for each category, which we will then study in more depth in Section VI.

A. Event-driven OSs

This is the most common approach for OSs initially developed to target the domain of WSNs, such as Contiki or TinyOS for instance. The key idea of this model is that all processing on the system is triggered by an (external) event, typically signaled by an interrupt. As a consequence the kernel is roughly equivalent to an infinite loop handling all occurring events within the same context. Such an event handler typically runs to completion. While this approach is efficient in terms of memory consumption and low complexity, it imposes some substantial constraints to the programmer e.g., not all programs are easily expressed as a finite state machine [40]. OSs that fall in this category include Contiki, TinyOS, and OpenWSN. Because of its wider deployment and use (to the best of our knowledge), Contiki is arguably a good representative of this category of OS.

B. Multi-Threading OSs

Multi-threading is the traditional approach for most modern OSs (e.g. Linux), whereby each thread runs in its own context and manages its own stack. With this approach, some scheduling has to perform context switching between the threads. Each process is handled in its own thread and can, in general, be interrupted at any point. Stack memory can usually not be shared between threads. Hence, a multi-threading OS usually introduces some memory overhead due to stack over-provisioning and runtime overhead due to context switching. Operating systems that fall in this category include RIOT, nuttX, eCos, or ChibiOS. Because of its stronger focus on IoT requirements (to the best of our knowledge), RIOT is arguably a good representative of this category of OS.

C. Pure RTOSs

An RTOS focuses primarily on the goal of fulfilling real-time guarantees, in an industrial/commercial context. In this context, formal verification, certification, and standardization are usually of crucial importance. To allow model checking and formal verification, the programming model used in such OSs typically imposes strict constraints for developers. These restrictions often makes the OS rather inflexible and porting to other hardware platforms may become rather difficult. Operating systems for IoT devices that fall in this category include FreeRTOS, eCos, RTEMS, ThreadX, and a collection of other commercial products (generally closed source). FreeRTOS is to the best of our knowledge the most prominent open source RTOS for IoT devices, due to its wider use in various environments.

VI. Case Studies

Our case studies cover a representative OS from each of the three categories described above. Each case study describes concisely the different properties of the operating systems, as listed in Section III.

A. Case Study: Contiki

Contiki was originally developed by Adam Dunkels in 2003 as an OS around the uIP stack, targeting resource-constrained embedded systems. Over time, Contiki evolved into a more general OS that is used in areas such as the IoT, wireless sensor networks, and even retro computing. It supports a wide range of resource-constrained devices, including 8-bit AVR platforms, 16- and 20-bit MSP430 platforms, and 32-bit ARM Cortex M3 platforms.

Contiki has a monolithic **architecture**, in which there is a core system and a set of processes that are combined into a single system image during compilation. At runtime, all processes share the same memory space and privileges with the core system. The **scheduling model** employed by Contiki is cooperative thread scheduling, which requires that Contiki processes explicitly yield control back to the scheduler. For **memory allocation**, Contiki is designed primarily for static allocation. It has a few libraries that simplify memory management, such as *memb* and *mmem*. However, we are also aware of third-party dynamic allocation modules for Contiki, which implement the standard C *malloc* API.

There are two different network stacks that can be used to enable Contiki devices with **network connectivity**: the uIP stack, which was first developed as a standalone stack and was merged into Contiki after version 0.9; and the more light-weight Rime stack, which is oriented toward sensor network applications. uIP supports multiple protocols, such as 6LoWPAN, IPv4, IPv6, IPv6 neighbor discovery, IPv6 multicasting, RPL, TCP, and UDP. The **network buffer management** is made through a separate module called Queuebuf, which allocates packet buffers from a static pool of memory.

The **programming model** is based on *Protothreads*, which is a sort of light-weight, cooperative threading concept similar to continuations [42]. The main **programming language** supported by Contiki is C, but there exist runtime environments

name	category	MCU w/o MMU	< 32 kB RAM	6LoWPAN	RTOS scheduler	HAL	energy-efficient MAC layers
Contiki	event-driven	✓	✓	✓	✘	✓	✓
RIOT	multi-threading	✓	✓	✓	✓	✓	✘[a]
FreeRTOS	RTOS	✓	✓	✘[b]	✓	✘	✘[c]
uClinux	multi-threading	✓	✘	✓	✘	✓	✘
Android	multi-threading	✘	✘	✘	✘	✓	✘
Arduino	other	✓	✓	✘	✘	✓[d]	✘

TABLE II: Key features of representatives of several categories. (✓) full support, (✘) no support. The table compares the OS for support of MCUs without MMU, MCUs with less than 100 kB of RAM, a 6LoWPAN network stack, a real-time capable scheduler, a hardware abstraction layer (HAL), and energy-efficient MAC layers.

[a] so far only as part of the OpenWSN stack, more implementations are planned
[b] available from third parties
[c] available from third parties
[d] limited portability

that enable development in languages such as Java [92] and Python [93]. Contiki provides a **hardware abstraction layer**, in which hardware-specific functionality is put in separate components, and each supported Contiki platform implements a common API for using that hardware. For instance, clocks, radio drivers, and sensors each have their own API that is implemented differently depending on the platform.

The **debugging facilities** available to Contiki developers consist primarily of the Cooja/MSPsim simulator, which combines network simulation with cycle-accurate emulation of the hardware platforms. This simulator contains standard debugging features such as setting breakpoints, reading from and writing to specific memory addresses, and single-stepping through instructions. Contiki also conveniently supports hardware used on several open testbeds, e.g., IoT-LAB [5] and Indriya [94].

Beside the networking capabilities and the core system functionality, Contiki has an ample **feature set**. It provides features such as a shell, a file system, a database management system, runtime dynamic linking, cryptography libraries, and a fine-grained power tracing tool. To enhance the quality of all these features, Contiki provides certain **testing** facilities, including unit testing, regression testing, and full system integration testing. Contiki code contributions are automatically tested with a test suite using Travis CI [95].

A number of **standards**—primarily related to networking—are supported by Contiki. For instance, Contiki implements several IETF standards for low-power IPv6 networking, including 6LoWPAN and RPL. Contiki has also gotten its core IPv6 functionality **certified** in the IPv6 Ready Logo Program, attaining a silver certification. The **documentation** of Contiki is of varying detail for different parts of the system. The source code is documented using the open-source Doxygen tool, and other things such as tutorials and high-level technical descriptions are provided through a project Wiki.

Today, the core parts of Contiki have reached a high **maturity of the code**, but there are also less used experimental parts of Contiki. The latter are primarily applications or libraries, which may have been developed as part of research projects. Multiple real-world deployments are based on Contiki, and it is widely used in commercial IoT products, as well as in academic research on WSN and other types of constrained wireless multi-hop networks. Along with TinyOS, Contiki has become one of the most well-known and widely used OSs for WSN.

Contiki is developed by a large community of professional developers, researchers, and hobbyists. The development is organized around a GitHub repository, through which anyone can submit *pull requests* containing code contributions. All source code in Contiki must have a 3-clause BSD **license**, or another license with similar terms. The source code is **maintained** by a merge team, which review incoming code contributions from the Contiki community, and make larger decisions such as architectural changes and release cycles. A number of source code forks exist, in which new features are developed by independent teams, or in which companies maintain their own versions of Contiki, possibly with support for their own hardware. Furthermore, because Contiki has many academic users, research projects are frequently developed for specific versions of Contiki. An active official mailing list for the project is the main source of support for most users.

B. Case Study: RIOT

RIOT development was launched in 2013, based on a microkernel architecture inherited from FireKernel [46], a microkernel initially developed for WSN scenarios with real-time requirements. RIOT development has so far focused on widening IoT hardware support (8, 16, 32 bit MCUs), efficient cross-platform code, and the development and maintenance of several networks stacks.

RIOT is based on a microkernel **architecture** with full multi-threading. Since multi-threading typically introduces

run-time as well as memory overhead, particular efforts were put into designing efficient context switching, IPC (blocking and non-blocking), and a small thread control block (TCB). As a result, context switching in RIOT is achieved in a small number of CPU cycles (e.g., less than 100 CPU cycles on an ARM platform when triggered from interrupt context) and the TCB is reduced to 46 bytes on 32 bit platforms, for instance. RIOT provides a tickless **scheduler** that works without any periodic events. Whenever there are no pending tasks, RIOT will switch to the *idle* thread, which can use the deepest possible sleep mode, depending on peripheral devices in use. Only interrupts (external or kernel-generated) wake up the system from idle state. RIOT supports both dynamic and static **memory allocation**. However, only static methods are used within the kernel, which enables RIOT to fulfill *deterministic* requirements, by enforcing constant periods for kernel tasks (e.g., scheduler run, inter-process communication, timer operations).

Several stacks are available in RIOT to support **network connectivity**. The *gnrc* network stack is based on standard IP protocols, supporting 6LoWPAN, IPv6, RPL (non-storing and storing mode), UDP, and CoAP, implemented in a modular fashion, leveraging generic, well-defined interfaces and IPC [35]. Other network stacks include for example *CCN-lite*, implementing the Information Centric Networking (ICN) paradigm, and *OpenWSN*, implementing the full 6TiSCH [26], [47] protocol suite, each available via BSD-like packages. The default network stack of RIOT is *gnrc*, within which packets, headers, and other networking meta data is stored in a centralized **network buffer** structure, whereby only pointers are being passed between the layers. Beside several network stacks, RIOT provides a wide range of diverse **features**, such as a shell, various crypto libraries, or sophisticated data structures.

The **programming model** in RIOT follows a classical multi-threading concept with a memory-passing IPC between threads. Its kernel is written in C (with minor parts being implemented in assembler). However, both C and C++ are available as **programming language** for applications and application libraries. RIOT has a well defined **hardware abstraction layer** for peripheral interfaces as well as for networking, sensor, and actuator devices. Leveraging the fact that RIOT is written in ANSI C, well-known, established **debugging** tools can be used, such as GDB, Valgrind etc. RIOT also provides a way to run instances of the OS as processes on Linux or Mac OS, which allows both easy debugging of embedded code, and virtual network emulation using either *nativenet* to emulate a single ethernet link, or the *desvirt* framework [96] for more complex topologies. Furthermore, Cooja can also be used to simulate platforms supported by this simulator. RIOT provides a set of unittests and applications for smoke and regression **testing**. Continuous integration testing is performed on the web-based service platform Travis. Additionally, a distributed test framework was designed, in order to conduct the tests on all supported platforms [36]. Tests can also be carried out on a number of open testbeds supported by RIOT e.g., IoT-LAB [97] [5] or DES-Testbed [98].

On the system side, RIOT focusses on implementing **standard** interfaces like POSIX. For the networking part, RIOT focusses on open standard protocols specified by bodies such as IETF, IRTF, W3C, OMA and the like. As an open source project stemming mostly from academia so far, no **certification** efforts have been conducted on the code base, to date. RIOT is driven by an open source community which strives to provide a comprehensive **documentation**, both on the API level as well as on the architectural level. While the interface and inline documentation of the code itself has already achieved a good standard (using Doxygen), example code and high-level descriptions are currently being overhauled by the community. Core parts of RIOT have been used for years by a community of users and developers – so far mostly academics, but recently also industry is using RIOT, primarily for more convenient prototyping. For instance, the kernel can be considered **mature**, as only minor bugs have been revealed during the last years. Other parts of RIOT (for example the network stack) are comparably younger and still subject to changes, e.g. as a result of co-evolving with new IoT protocol standards, as they appear. However, the use of standard and generic interfaces (such as POSIX sockets or *netapi* [35]) are stabilizing the usage of the code base.

The source code is openly available and **licensed** under LGPL. RIOT's master branch on GitHub is currently **maintained** by several tens of developers that are in charge of reviewing and merging external contributions that are provided through *pull requests*. A lively official mailing list is also used by the community to discuss various technical and community-related matters.

C. Case Study: FreeRTOS

Originally developed by Richard Barry in 2002, FreeRTOS is now maintained and distributed by Real Time Engineers Ltd. FreeRTOS is deployed in various industrial/commercial environments, and is the base of several research projects. In contrast to many other RTOSs, FreeRTOS is designed to be small, simple, portable, and easy to use. Therefore, it is supported by a large **community** and has been ported to a big number of MCUs, including hardware available on open testbeds (e.g. IoT-LAB [5]). There are several forks of the FreeRTOS code-base available: for instance, SafeRTOS (focusing on safety) and OpenRTOS (removing all all references to GPL).

FreeRTOS itself implements a fairly simple **architecture**, as it comprises of only four C files and is more a threading library than a full-fledged operating system. The only provided functionalities are thread handling, mutexes, semaphores, and software timers. In the default configuration FreeRTOS uses a preemptive, priority based round-robin **scheduler**, which is triggered by a periodic timer tick interrupt. Since version 7.3.0 (released October 31 2012) the scheduler further supports a tickless mode. In order to fulfill real-time guarantees, it is ensured that FreeRTOS uses only deterministic operations from inside a critical section or interrupt. In FreeRTOS, queues are used for IPC which support blocking and non-blocking insert (using deep copy), as well as remove functions.

FreeRTOS defines five different memory allocation schemes: (i) allocate only, (ii) allocate and free with a simplistic, fast algorithm, (iii) wrapping C library `malloc()` and `free()` for thread safety, (iv) a more complex but fast allocate and free algorithm with memory coalescence, and (v) a more advanced version of (iv) that allows to span the heap over several memory sections.

FreeRTOS itself does not provide any **networking capabilities**. However, many additional tools and libraries are available in the FreeRTOS ecosystem (mostly through third-parties). Most notable Real Time Engineers Ltd. offers an official *FreeRTOS+TCP* add-on supporting an Ethernet-based IPv4 stack with support for UDP, TCP and supporting protocols. Furthermore ports of third-party embedded network stacks as lwIP [99] or older versions of Nanostack [100] are available. The network **buffer management** depends on the stack used. The official *FreeRTOS+TCP* for example can be configured to use a statically pre-allocated buffer or to allocate buffer space dynamically on-demand.

FreeRTOS supports a multi-threading **programming model** with statically instantiated tasks. The **programming language** used for the OS itself is C, which enables users to integrate it seamlessly also in any C++ application. As stated above, the **feature set** of the basic system is limited to scheduling, threading and SW timers. FreeRTOS does not define a portable **driver model** or MCU **peripheral abstraction** interfaces. Instead it works together with vendor supplied board support packages. For **testing** and **debugging** the system also depends on third-party solutions, though the design of the OS makes it possible to be integrated in most existing development processes.

In order to comply with regulations and requirements of industrial use cases, FreeRTOS emphasizes on strict coding standards, quality management, and **certification**. Consequently, FreeRTOS has become part of various formal verification efforts [101], [102]. SafeRTOS has been certified by TÜV SÜD as IEC 61508 compliant and against the EN 62304 and FDA 540(k) regulatory requirements. Real Time Engineers Ltd. provides extensive **documentation** in terms of books, trainings, and commercial support. The code is **licensed** under GPL with an optional linking exception that allows to link proprietary, closed source code.

VII. Conclusion

In this survey, we have analyzed the various requirements that should be fulfilled by an OS for low-end IoT devices, which are too resource-constrained to run traditional operating systems such as Linux. We have overviewed key aspects for such an OS, both from technical and non-technical points of view. Considering these aspects, we have then surveyed available OSs that could qualify to become the go-to OS for IoT devices.

We have mostly focused on open source operating systems because, in the context of IoT, acute privacy and security concerns are to be anticipated. Such concerns present an immense challenge that is easier to address with open source code, which offers higher potential for transparency, trustworthiness, and security. We note that, in order to benefit fully from the advantages of open source in terms of trustworthiness, it is also necessary to use open source toolchains to produce and deploy binaries on IoT devices (and to rule out dependency on untrusted third-party servers/cloud services to produce and deploy these binaries). In the long run, the collaborative nature of most open source development increases the probability that bugs are found and fits better the needs of SMEs. According to recent studies [103], such companies will be driving IoT innovation in the near future, but are more likely than bigger companies to need IoT software development and maintenance costs sharing.

In this survey, we have identified three categories of OSs, within which some have the potential to become the equivalent to Linux in the IoT. Multi-threaded OSs are technically closest to Linux, and within this category, RIOT is currently the most prominent open source OS. Event-driven OSs use a different programming paradigm to fit on devices with even less resources, and within this category, Contiki is currently the most prominent open source OS. RTOSs focus on guarantees for worst-case execution times and worst-case interrupt latency. In this category, FreeRTOS is currently the most prominent open source OS.

Our conclusions are that there are a plethora of different OSs for the IoT, allowing users to select an OS that fits their criteria best. Our survey has covered many of the trade-offs being made by system designers regarding the requirements and constraints of current IoT applications and hardware platforms. As the IoT field is developing at a rapid pace, however, the final word is yet to be made regarding what type of architecture and capabilities an ideal OS for the IoT should have.

Acknowledgements – The authors wish to thank Simon Duquennoy, Thiemo Voigt, and Björn Lichtblau for their useful comments and suggestions. This work was partly financed by ANR within SAFEST (ANR- 11-SECU-004), by SSF, and by VINNOVA.

References

[1] E. Upton and G. Halfacree, *Meet the Raspberry Pi*. John Wiley & Sons, 2012.
[2] Arduino Due. [Online]. Available: http://arduino.cc/en/Main/arduinoBoardDue
[3] Redwire Llc. Redwire Econotag II. [Online]. Available: http://redwire.myshopify.com/products/econotag-ii
[4] Zolertia. Z1 Datasheet. [Online]. Available: http://www.zolertia.com/
[5] IoT-LAB: Very large scale open wireless sensor network testbed. [Online]. Available: https://www.iot-lab.info/hardware/m3/
[6] OpenMote. OpenMote-CC2538. [Online]. Available: http://www.openmote.com/hardware/openmote-cc2538-en.html
[7] MoteIV Corporation. Telos – Ultra Low Power IEEE 802.15.4 Compliant Wireless Sensor Module, Datasheet. [Online]. Available: http://www.willow.co.uk/html/telosb_mote_platform.php
[8] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained node networks," RFC 7228 (Informational), Internet Engineering Task Force, May 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7228.txt
[9] M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne *et al.*, "Making sensor networks ipv6 ready," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, 2008, pp. 421–422.
[10] Pebble Smart Watch. [Online]. Available: https://getpebble.com

[11] Huawei. Huawei Network Congress 2015 Announcement. [Online]. Available: http://pr.huawei.com/en/news/hw-432402-agilenetwork3.0.htm

[12] mbed OS. [Online]. Available: https://mbed.org/technology/os/

[13] Anand Karwa, Trak.In. Google Brillo – An Internet Of Things OS That Runs on 32 MB RAM. [Online]. Available: http://trak.in/tags/business/2015/05/23/google-brillo-internet-of-things-operating-system/

[14] L. Mirani, "Chip-makers are Betting that Moore's Law Won't Matter in the Internet of Things," 2014. [Online]. Available: http://qz.com/218514

[15] M. Dohler, T. Watteyne, T. Winter, and D. Barthel, "Routing Requirements for Urban Low-Power and Lossy Networks," RFC 5548 (Informational), Internet Engineering Task Force, May 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5548.txt

[16] J. Martocci, P. D. Mil, N. Riou, and W. Vermeylen, "Building Automation Routing Requirements in Low-Power and Lossy Networks," RFC 5867 (Informational), Internet Engineering Task Force, Jun. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5867.txt

[17] K. Pister, P. Thubert, S. Dwars, and T. Phinney, "Industrial Routing Requirements in Low-Power and Lossy Networks," RFC 5673 (Informational), Internet Engineering Task Force, Oct. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5673.txt

[18] A. Brandt, J. Buron, and G. Porcu, "Home Automation Routing Requirements in Low-Power and Lossy Networks," RFC 5826 (Informational), Internet Engineering Task Force, Apr. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5826.txt

[19] Rose, Karen and Eldridge, Scott and Chapin, Lyman, *The Internet of Things: An Overview*. The Internet Society (ISOC), 2015.

[20] Freescale. The MC13224V SoC. [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC13224V

[21] ST Microelectronics. STM32F100VC Microcontroller. [Online]. Available: http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1031/LN775/PF216851

[22] D. Dietrich, D. Bruckner, G. Zucker, and P. Palensky, "Communication and computation in buildings: A short introduction and overview," *Industrial Electronics, IEEE Transactions on*, vol. 57, no. 11, pp. 3577–3584, Nov 2010.

[23] R. Jedermann, T. Pötsch, and C. Lloyd, "Communication techniques and challenges for wireless food quality monitoring," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 372, no. 2017, 2014.

[24] W. Dong, C. Chen, X. Liu, and J. Bu, "Providing os support for wireless sensor networks: challenges and approaches," *Communications Surveys & Tutorials, IEEE*, vol. 12, no. 4, pp. 519–530, 2010.

[25] L. Saraswat and P. S. Yadav, "A comparative analysis of wireless sensor network operating systems," *International Journal of Engineering and Technoscience*, vol. 1, no. 1, pp. 41–47, 2010.

[26] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, "Standardized protocol stack for the internet of (important) things," *Communications Surveys & Tutorials, IEEE*, vol. 15, no. 3, pp. 1389–1406, 2013.

[27] R. Min, M. Bhardwaj, S.-H. Cho, N. Ickes, E. Shih, A. Sinha, A. Wang, and A. Chandrakasan, "Energy-centric enabling technologies for wireless sensor networks," *IEEE wireless communications*, vol. 9, no. 4, pp. 28–39, 2002.

[28] A. Milenković, C. Otto, and E. Jovanov, "Wireless sensor networks for personal health monitoring: Issues and an implementation," *Computer communications*, vol. 29, no. 13, pp. 2521–2533, 2006.

[29] B. Hughes, R. Meier, R. Cunningham, and V. Cahill, "Towards real-time middleware for vehicular ad hoc networks," in *Proceedings of the 1st ACM International Workshop on Vehicular Ad Hoc Networks*, ser. VANET '04. New York, NY, USA: ACM, 2004, pp. 95–96. [Online]. Available: http://doi.acm.org/10.1145/1023875.1023894

[30] ITRON project archive. [Online]. Available: http://www.ertl.jp/ITRON/home-e.html

[31] K. A. Stouffer, J. A. Falco, and K. A. Scarfone, "Sp 800-82. guide to industrial control systems (ics) security: Supervisory control and data acquisition (scada) systems, distributed control systems (dcs), and other control system configurations such as programmable logic controllers (plc)," Gaithersburg, MD, United States, Tech. Rep., 2011.

[32] J.-H. Hoepman and B. Jacobs, "Increased security through open source," *Commun. ACM*, vol. 50, no. 1, pp. 79–83, Jan. 2007. [Online]. Available: http://doi.acm.org/10.1145/1188913.1188921

[33] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*. IEEE, 2004, pp. 79–88.

[34] A. Castellani, G. Ministeri, M. Rotoloni, L. Vangelista, and M. Zorzi, "Interoperable and globally interconnected Smart Grid using IPv6 and 6LoWPAN," in *Communications (ICC), 2012 IEEE International Conference on*, June 2012, pp. 6473–6478.

[35] H. Petersen, M. Lenders, M. Wählisch, O. Hahm, and E. Baccelli, "Old Wine in New Skins? Revisiting the Software Architecture for IP Network Stacks on Constrained IoT Devices," in *ACM MobiSys Workshop on IoT Challenges in Mobile and Industrial Systems (IoT-Sys)*, May 2015.

[36] P. Rosenkranz, M. Wählisch, E. Baccelli, and L. Ortmann, "A Distributed Test System Architecture for Open-source IoT Software," in *ACM MobiSys Workshop on IoT Challenges in Mobile and Industrial Systems (IoT-Sys)*, May 2015.

[37] J. Eriksson, A. Dunkels, N. Finne, F. Osterlind, and T. Voigt, "Mspsim–an extensible simulator for msp430-equipped sensor boards," in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, 2007, p. 27.

[38] Emul8. [Online]. Available: http://emul8.org/

[39] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, 2008.

[40] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors." in *LCN*. IEEE Computer Society, 2004, pp. 455–462. [Online]. Available: http://dblp.uni-trier.de/db/conf/lcn/lcn2004.html#DunkelsGV04

[41] Contiki Operating System. http://www.contiki-os.org.

[42] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Boulder, Colorado, USA, Nov. 2006.

[43] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *32nd IEEE INFOCOM*, IEEE. Turin, Italy: IEEE, 2013.

[44] E. Baccelli, O. Hahm, H. Petersen, and K. Schleiser, "RIOT and the Evolution of IoT Operating Systems and Applications," *ERCIM News*, vol. 2015, no. 101, 2015. [Online]. Available: http://ercim-news.ercim.eu/en101/special/riot-and-the-evolution-of-iot-operating-systems-and-applications

[45] RIOT Operating System. http://www.riot-os.org.

[46] H. Will, K. Schleiser, and J. H. Schiller, "A real-time kernel for wireless sensor networks employed in rescue scenarios," in *IEEE LCN*, 2009.

[47] Berkeley's OpenWSN Project. [Online]. Available: http://openwsn.berkeley.edu/

[48] "CCN Lite: Lightweight implementation of the Content Centric Networking protocol," 2014. [Online]. Available: http://ccn-lite.net

[49] R. Barry. FreeRTOS, a FREE open source RTOS for small embedded real time systems. http://www.freertos.org.

[50] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An Operating System for Sensor Networks," in *Ambient Intelligence*, W. Weber, J. M. Rabaey, and E. Aarts, Eds. Berlin Heidelberg: Springer-Verlag, 2005, ch. 7, pp. 115–148. [Online]. Available: http://dx.doi.org/10.1007/3-540-27139-2_7

[51] P. Levis, "Experiences from a Decade of TinyOS Development," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 207–220. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387901

[52] Contributors of nuttx.org, "NuttX Real-Time Operating System — NuttX Real-Time Operating System," 2015, [Online; accessed 10-March-2015]. [Online]. Available: http://nuttx.org

[53] eCos Embedded Operating System. [Online]. Available: http://ecos.sourceware.org

[54] Home of the L4 community. [Online]. Available: http://l4hq.org/

[55] H. Härtig and M. Roitzsch, "Ten years of research on L4-based real-time systems," in *Proceedings of the 8th Real-Time Linux Workshop*, 2006.

[56] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 207–220.

[57] Embedded Linux/Microcontroller Project. [Online]. Available: http://www.uclinux.org

[58] Google, Open Handset Alliance. Android Operating System. [Online]. Available: https://www.android.com/

[59] Google. Project Brillo. [Online]. Available: https://developers.google.com/brillo/

[60] ChibiOS/RT. [Online]. Available: http://www.chibios.org

[61] CooCox CoOS. [Online]. Available: http://www1.coocox.org/CoOS.htm

[62] ERIKA Enterprise. [Online]. Available: http://erika.tuxfamily.org/drupal/

[63] G. Strazdins, A. Elsts, and L. Selavo, "Mansos: easy to use, portable and resource efficient operating system for networked embedded devices," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2010, pp. 427–428.

[64] S. C. Kim, H. Kim, J. Song, M. Yu, and P. Mah, "NanoQplus: A Multi-Threaded Operating System with Memory Protection Mechanism for WSNs," *Proceedings of the CKWSN*, vol. 20, no. 08, 2008.

[65] NanoRK Operating System. [Online]. Available: http://www.nanork.org

[66] Nut/OS. [Online]. Available: http://www.ethernut.de/en/software/

[67] Liquorice OS for Embedded Systems. [Online]. Available: http://sourceforge.net/p/liquorice/wiki/Home/

[68] RTEMS – Real-Time Executive for Multiprocessor Systems. [Online]. Available: http://www.rtems.org

[69] SOS 2.X. [Online]. Available: https://projects.nesl.ucla.edu/public/sos-2x

[70] S. Bhatti et al., "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Networks and Applications*, 2005.

[71] MANTIS OS. [Online]. Available: http://www.sourceforge.net/projects/mantisos/

[72] Lorien. [Online]. Available: http://lorienos.sourceforge.net/index.php/Main_Page

[73] LiteOS. [Online]. Available: http://www.liteos.net/

[74] Express Logic, Inc. ThreadX. [Online]. Available: http://rtos.com/products/threadx/

[75] ——. Express Logic's ThreadX® To Bring Full RTOS Capabilities to the ARM® mbed™ Ecosystem. [Online]. Available: http://rtos.com/news/detail/express_logics_threadx_brings_full_rtos_capabilities_to_arms_mbedos_ec/

[76] Blackberry Ltd. QNX. [Online]. Available: http://www.qnx.com/

[77] Wind River Systems. VxWorks. [Online]. Available: http://www.windriver.com/products/vxworks/

[78] Wind River Rocket. [Online]. Available: http://www.windriver.com/products/operating-systems/rocket/

[79] SYSGO. PikeOS. [Online]. Available: http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/

[80] Segger. embOS. [Online]. Available: https://www.segger.com/embos.html

[81] Mentor Graphics. Nucleus RTOS. [Online]. Available: http://www.mentor.com/embedded-software/nucleus/

[82] SCIOPTA Systems AG. SCIOPTA. [Online]. Available: http://www.sciopta.com/products/kernel.html

[83] Micrium. uC/OS-II. [Online]. Available: http://micrium.com/rtos/ucosii/overview/

[84] ——. uC/OS-III. [Online]. Available: http://micrium.com/rtos/ucosiii/overview/

[85] Green Hills Software. µ-velOSity. [Online]. Available: http://www.ghs.com/products/micro_velosity.html

[86] Microsoft. Windows CE. [Online]. Available: http://microsoft.com/windowsce/

[87] Huawei. LiteOS. [Online]. Available: https://github.com/OIOTC/Liteos

[88] Arduino. [Online]. Available: http://arduino.cc/

[89] Espruino – JavaScript for Microcontrollers. [Online]. Available: http://www.espruino.com/

[90] Tessel Embedded Development Platform. [Online]. Available: https://tessel.io/

[91] node OS. [Online]. Available: http://node-os.com/

[92] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, a feature-rich vm for the resource poor," in *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Berkeley, CA, USA, 2009.

[93] S. Bocchino, S. Fedor, and M. Petracca, "PyFUNS: A Python Framework for Ubiquitous Networked Sensors," in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN)*, Porto, Portugal, Feb. 2015.

[94] M. Doddavenkatappa, M. C. Chan, and A. Ananda, "Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed," in *Proceedings of the Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities (TridentCom)*, 2011.

[95] Travis CI. [Online]. Available: https://travis-ci.org

[96] The DES Testbed virtualization framework. [Online]. Available: https://github.com/des-testbed/desvirt

[97] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed," in *Proceedings of the 2nd IEEE World Forum on Internet of Things (WF-IoT)*, December 2015.

[98] M. Günes, B. Blywis, M. Frey, O. Hahm, F. Juraschek, P. Kumar, Q. Mushtaq, and K. Schleiser. (2008-2014) DES-Testbed. http://www.des-testbed.net. Homepage of the DES-Testbed. [Online]. Available: http://www.des-testbed.net

[99] lwIP – A Lightweight TCP/IP Stack. [Online]. Available: http://savannah.nongnu.org/projects/lwip/

[100] Sensinode Ldt., "NanoStack Manual," 2007.

[101] D. Déharbe, S. Galvão, and A. Moreira, "Formalizing FreeRTOS: First Steps," in *Formal Methods: Foundations and Applications*, ser. Lecture Notes in Computer Science, M. Oliveira and J. Woodcock, Eds. Springer Berlin Heidelberg, 2009, vol. 5902, pp. 101–117. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10452-7_8

[102] C. Pronk, "Verifying FreeRTOS; a feasibility study," Delft University of Technology, Software Engineering Research Group, Tech. Rep., 2010.

[103] P. Basiliere and J. Tully, *Gartner Study: Makers and Startups Are the Ones Shaping the Internet of Things*. Maverick Research, Gartner, 2014.